12 Understanding the ASP.NET Core platform

This chapter covers

- Understanding the basic structure of an ASP.NET Core application
- Understanding the HTTP request processing pipeline and middleware components
- Creating custom middleware components

The ASP.NET Core platform is the foundation for creating web applications; it provides the features that make it possible to use frameworks like MVC and Blazor. In this chapter, I explain how the basic ASP.NET Core features work, describe the purpose of the files in an ASP.NET Core project, and explain how the ASP.NET Core request pipeline is used to process HTTP requests and demonstrate the different ways that it can be customized.

Don't worry if not everything in this chapter makes immediate sense or appears to apply to the applications you intend to create. The features I describe in this chapter are the underpinnings for everything that ASP.NET Core does, and understanding how they work helps provide a context for understanding the features that you will use daily, as well as giving you the knowledge you need to diagnose problems when you don't get the behavior you expect. Table 12.1 puts the ASP.NET Core platform in context.

Table 12.1 Putting the ASP.NET Core platform in context

Question	Answer
What is it?	The ASP.NET Core platform is the foundation on which web applications are built and provides features for processing HTTP requests.
Why is it useful?	The ASP.NET Core platform takes care of the low-level details of web applications so that developers can focus on features for the end user.
How is it used?	The key building blocks are services and middleware components, both of which can be created using top-level statements in the `Program.cs` file.
Are there any pitfalls or limitations?	The use of the `Program.cs` file can be confusing, and close attention must be paid to the order of the statements it contains.
Are there any alternatives?	The ASP.NET Core platform is required for ASP.NET Core applications, but you can choose not to work with the platform directly and rely on just the higher-level ASP.NET Core features, which are described in later chapters.

Table 12.2 provides a guide to the chapter.

Table 12.2 Chapter guide

Problem	Solution	Listing
Creating a middleware component	Call the `Use` or `UseMiddleware` method to add a function or class to the request pipeline.	6–8
Modifying a response	Write a middleware component that uses the return pipeline path.	9
Preventing other components from processing a request	Short-circuit the request pipeline or create terminal middleware.	10, 12–14
Using different sets of middleware	Create a pipeline branch.	11
Configuring middleware components	Use the options pattern.	15–18

12.1 Preparing for this chapter

To prepare for this chapter, I am going to create a new project named Platform, using the template that provides the minimal ASP.NET Core setup. Open a new PowerShell command prompt from the Windows Start menu and run the commands shown in listing 12.1.

TIP You can download the example project for this chapter—and for all the other chapters in this book—from https://github.com/manningbooks/pro-asp.net-core-7. See chapter 1 for how to get help if you have problems running the examples.

Listing 12.1 Creating the project

```
dotnet new globaljson --sdk-version 7.0.100 --output Platform
dotnet new web --no-https --output Platform --framework net7.0
dotnet new sln -o Platform
dotnet sln Platform add Platform
```

If you are using Visual Studio, open the `Platform.sln` file in the `Platform` folder. If you are using Visual Studio Code, open the `Platform` folder. Click the Yes button when prompted to add the assets required for building and debugging the project.

Open the `launchSettings.json` file in the `Properties` folder and change the ports that will be used to handle HTTP requests, as shown in listing 12.2.

Listing 12.2 Setting ports in the launchSettings.json file in the Properties folder

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:5000",
      "sslPort": 0
    }
  },
  "profiles": {
    "Platform": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

12.1.1 Running the example application

To start the application, run the command shown in listing 12.3 in the `Platform` folder.

Listing 12.3 Starting the example application

```
dotnet run
```

Open a new browser window and use it to request http://localhost:5000. You will see the output shown in figure 12.1.

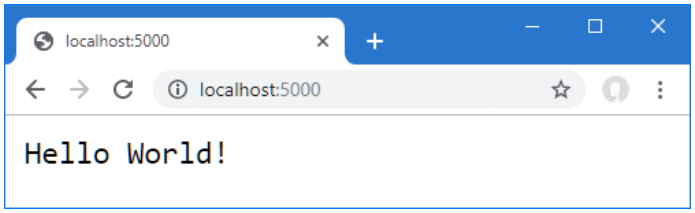

Figure 12.1 Running the example application

12.2 Understanding the ASP.NET Core platform

To understand ASP.NET Core, it is helpful to focus on just the key features: the request pipeline, middleware, and services. Understanding how these features fit together—even without going into detail—provides useful context for understanding the contents of the ASP.NET Core project and the shape of the ASP.NET Core platform.

12.2.1 Understanding middleware and the request pipeline

The purpose of the ASP.NET Core platform is to receive HTTP requests and send responses to them, which ASP.NET Core delegates to *middleware components*. Middleware components are arranged in a chain, known as the *request pipeline*.

When a new HTTP request arrives, the ASP.NET Core platform creates an object that describes it and a corresponding object that describes the response that will be sent in return. These objects are passed to the first middleware component in the chain, which inspects the request and modifies the response. The request is then passed to the next middleware component in the chain, with each component inspecting the request and adding to the response. Once the request has made its way through the pipeline, the ASP.NET Core platform sends the response, as illustrated in figure 12.2.

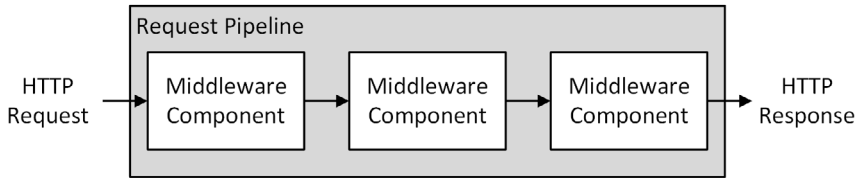

Figure 12.2 The ASP.NET Core request pipeline

Some components focus on generating responses for requests, but others are there to provide supporting features, such as formatting specific data types or reading and writing cookies. ASP.NET Core includes middleware components that solve common problems, as described in chapters 15 and 16, and I show how to create custom middleware components later in this chapter. If no response is generated by the middleware components, then ASP.NET Core will return a response with the HTTP 404 Not Found status code.

12.2.2 Understanding services

Services are objects that provide features in a web application. Any class can be used as a service, and there are no restrictions on the features that services provide. What makes services special is that they are managed by ASP.NET Core, and a feature called *dependency injection* makes it possible to easily access services anywhere in the application, including middleware components.

Dependency injection can be a difficult topic to understand, and I describe it in detail in chapter 14. For now, it is enough to know that there are objects that are managed by the ASP.NET Core platform that can be shared by middleware components, either to coordinate between components or to avoid duplicating common features, such as logging or loading configuration data, as shown in figure 12.3.

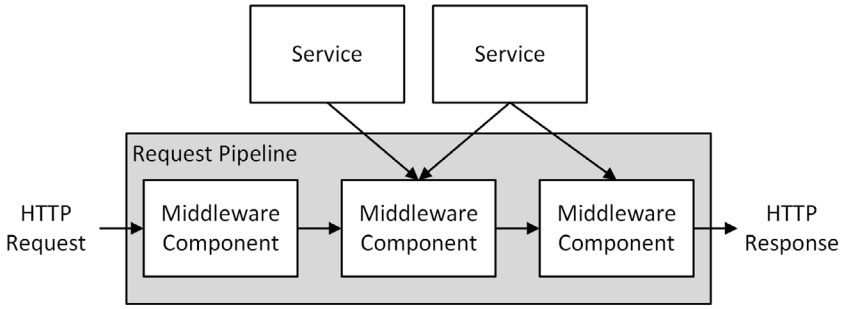

Figure 12.3 Services in the ASP.NET Core platform

As the figure shows, middleware components use only the services they require to do their work. As you will learn in later chapters, ASP.NET Core provides some basic services that can be supplemented by additional services that are specific to an application.

12.3 Understanding the ASP.NET Core project

The `web` template produces a project with just enough code and configuration to start the ASP.NET Core runtime with some basic services and middleware components. Figure 12.4 shows the files added to the project by the template.

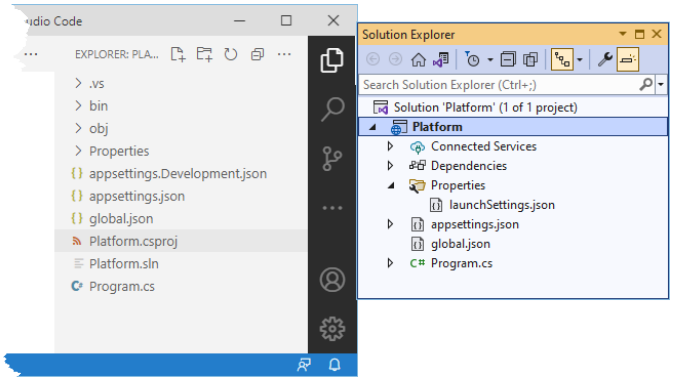

Figure 12.4 The files in the example project

Visual Studio and Visual Studio Code take different approaches to displaying files and folders. Visual Studio hides items that are not commonly used by the developer and nests related items together, while Visual Studio Code shows everything.

This is why the two project views shown in the figure are different: Visual Studio has hidden the `bin` and `obj` folders and nested the `appsettings.Development.json` file within the `appsettings.json` file. The buttons at the top of the Solution Explorer window can be used to prevent nesting and to show all the files in the project.

Although there are few files in the project, they underpin ASP.NET Core development and are described in table 12.3.

Table 12.3 The files and folders in the Example project

Name	Description
`appsettings.json`	This file is used to configure the application, as described in chapter 15.
`appsettings.Development.json`	This file is used to define configuration settings that are specific to development, as explained in chapter 15.
`bin`	This folder contains the compiled application files. Visual Studio hides this folder.
`global.json`	This file is used to select a specific version of the .NET Core SDK.
`Properties/launchSettings.json`	This file is used to configure the application when it starts.
`obj`	This folder contains the intermediate output from the compiler. Visual Studio hides this folder.
`Platform.csproj`	This file describes the project to the .NET Core tools, including the package dependencies and build instructions, as described in the "Understanding the Project File" section. Visual Studio hides this file, but it can be edited by right-clicking the project item in the Solution Explorer and selecting Edit Project File from the pop-up menu.
`Platform.sln`	This file is used to organize projects. Visual Studio hides this folder.
`Program.cs`	This file is the entry point for the ASP.NET Core platform and is used to configure the platform, as described in the "Understanding the Entry Point" section

12.3.1 Understanding the entry point

The `Program.cs` file contains the code statements that are executed when the application is started and that are used to configure the ASP.NET platform and the individual frameworks it supports. Here is the content of the `Program.cs` file in the example project:

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

This file contains only top-level statements. The first statement calls the `WebApplication.CreateBuilder` and assigns the result to a variable named `builder`:

```
...
var builder = WebApplication.CreateBuilder(args);
...
```

This method is responsible for setting up the basic features of the ASP.NET Core platform, including creating services responsible for configuration data and logging, both of which are described in chapter 15. This method also sets up the HTTP server, named Kestrel, that is used to receive HTTP requests.

The result from the `CreateBuilder` method is a `WebApplicationBuilder` object, which is used to register additional services, although none are defined at present. The `WebApplicationBuilder` class defines a `Build` method that is used to finalize the initial setup:

```
...
var app = builder.Build();
...
```

The result of the `Build` method is a `WebApplication` object, which is used to set up middleware components. The template has set up one middleware component, using the `MapGet` extension method:

```
...
app.MapGet("/", () => "Hello World!");
...
```

`MapGet` is an extension method for the `IEndpointRouteBuilder` interface, which is implemented by the `WebApplication` class, and which sets up a function that will handle HTTP requests with a specified URL path. In this case, the function responds to requests for the default URL path, which is denoted by `/`, and the function responds to all requests by returning a simple `string` response, which is how the output shown in figure 12.1 was produced.

Most projects need a more sophisticated set of responses, and Microsoft provides middleware as part of ASP.NET Core that deals with the most common features required by web applications, which I describe in chapters 15 and 16. You can also create your own middleware, as described in the "Creating Custom Middleware" section, when the built-in features don't suit your requirements.

The final statement in the `Program.cs` file calls the `Run` method defined by the `WebApplication` class, which starts listening to HTTP requests.

Even though the function used with the `MapGet` method returns a string, ASP.NET Core is clever enough to create a valid HTTP response that will be understood by browsers. While ASP.NET Core is still running, open a new PowerShell command prompt and run the command shown in listing 12.4 to send an HTTP request to the ASP.NET Core server.

Listing 12.4 Sending an HTTP Request

```
(Invoke-WebRequest http://localhost:5000).RawContent
```

The output from this command shows that the response sent by ASP.NET Core contains an HTTP status code and the set of basic headers, like this:

```
HTTP/1.1 200 OK
Transfer-Encoding: chunked
Content-Type: text/plain; charset=utf-8
Date: Wed, 14 Dec 2022 08:09:13 GMT
Server: Kestrel
```

12.3.2 Understanding the project file

The `Platform.csproj` file, known as the *project file*, contains the information that .NET Core uses to build the project and keep track of dependencies. Here is the content that was added to the file by the Empty template when the project was created:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net7.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

</Project>
```

The `csproj` file is hidden when using Visual Studio; you can edit it by right-clicking the Platform project item in the Solution Explorer and selecting Edit Project File from the pop-up menu.

The project file contains XML elements that describe the project to MSBuild, the Microsoft build engine. MSBuild can be used to create complex build processes and is described in detail at https://docs.microsoft.com/en-us/visualstudio/msbuild/msbuild.

There is no need to edit the project file directly in most projects. The most common change to the file is to add dependencies on other .NET packages, but these are typically added using the command-line tools or the interface provided by Visual Studio.

To add a package to the project using the command line, open a new PowerShell command prompt, navigate to the `Platform` project folder (the one that contains the `csproj` file), and run the command shown in listing 12.5.

Listing 12.5 Adding a package to the project

```
dotnet add package Swashbuckle.AspNetCore --version 6.4.0
```

This command adds the `Swashbuckle.AspNetCore` package to the project. You will see this package used in chapter 20, but for now, it is the effect of the `dotnet add package` command that is important.

The new dependency will be shown in the `Platform.csproj` file:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net7.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Swashbuckle.AspNetCore" Version="6.4.0" />
  </ItemGroup>

</Project>
```

12.4 Creating custom middleware

As mentioned, Microsoft provides various middleware components for ASP.NET Core that handle the features most commonly required by web applications. You can also create your own middleware, which is a useful way to understand how ASP.NET Core works, even if you use only the standard components in your projects. The key method for creating middleware is `Use`, as shown in listing 12.6.

Listing 12.6 Creating custom middleware in the Program.cs file in the Platform folder

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.Use(async (context, next) => {
    if (context.Request.Method == HttpMethods.Get
            && context.Request.Query["custom"] == "true") {
        context.Response.ContentType = "text/plain";
        await context.Response.WriteAsync("Custom Middleware \n");
    }
    await next();
});

app.MapGet("/", () => "Hello World!");

app.Run();
```

The `Use` method registers a middleware component that is typically expressed as a lambda function that receives each request as it passes through the pipeline (there is another method used for classes, as described in the next section).

The arguments to the lambda function are an `HttpContext` object and a function that is invoked to tell ASP.NET Core to pass the request to the next middleware component in the pipeline.

The `HttpContext` object describes the HTTP request and the HTTP response and provides additional context, including details of the user associated with the request. Table 12.4 describes the most useful members provided by the `HttpContext` class, which is defined in the `Microsoft.AspNetCore.Http` namespace.

Table 12.4 Useful HttpContext members

Name	Description
`Connection`	This property returns a `ConnectionInfo` object that provides information about the network connection underlying the HTTP request, including details of local and remote IP addresses and ports.
`Request`	This property returns an `HttpRequest` object that describes the HTTP request being processed.
`RequestServices`	This property provides access to the services available for the request, as described in chapter 14.
`Response`	This property returns an `HttpResponse` object that is used to create a response to the HTTP request.
`Session`	This property returns the session data associated with the request. The session data feature is described in chapter 16.
`User`	This property returns details of the user associated with the request, as described in chapters 37 and 38.
`Features`	This property provides access to request features, which allow access to the low-level aspects of request handling. See chapter 16 for an example of using a request feature.

The ASP.NET Core platform is responsible for processing the HTTP request to create the `HttpRequest` object, which means that middleware and endpoints don't have to worry about the raw request data. Table 12.5 describes the most useful members of the `HttpRequest` class.

Table 12.5 Useful HttpRequest members

Name	Description
`Body`	This property returns a stream that can be used to read the request body.
`ContentLength`	This property returns the value of the `Content-Length` header.
`ContentType`	This property returns the value of the `Content-Type` header.
`Cookies`	This property returns the request cookies.
`Form`	This property returns a representation of the request body as a form.
`Headers`	This property returns the request headers.
`IsHttps`	This property returns `true` if the request was made using HTTPS.
`Method`	This property returns the HTTP verb—also known as the HTTP method—used for the request.
`Path`	This property returns the path section of the request URL.
`Query`	This property returns the query string section of the request URL as key-value pairs.

The `HttpResponse` object describes the HTTP response that will be sent back to the client when the request has made its way through the pipeline. Table 12.6 describes the most useful members of the `HttpResponse` class. The ASP.NET Core platform makes dealing with responses as easy as possible, sets headers automatically, and makes it easy to send content to the client.

Table 12.6 Useful HttpResponse members

Name	Description
`ContentLength`	This property sets the value of the `Content-Length` header.
`ContentType`	This property sets the value of the `Content-Type` header.
`Cookies`	This property allows cookies to be associated with the response.
`HasStarted`	This property returns `true` if ASP.NET Core has started to send the response headers to the client, after which it is not possible to make changes to the status code or headers.
`Headers`	This property allows the response headers to be set.
`StatusCode`	This property sets the status code for the response.
`WriteAsync(data)`	This asynchronous method writes a data string to the response body.
`Redirect(url)`	This method sends a redirection response.

When creating custom middleware, the `HttpContext`, `HttpRequest`, and `HttpResponse` objects are used directly, but, as you will learn in later chapters, this isn't usually required when using the higher-level ASP.NET Core features such as the MVC Framework and Razor Pages.

The middleware function I defined in listing 12.6 uses the `HttpRequest` object to check the HTTP method and query string to identify GET requests that have a `custom` parameter in the query string whose value is `true`, like this:

```
...
if (context.Request.Method == HttpMethods.Get
    && context.Request.Query["custom"] == "true") {
...
```

The `HttpMethods` class defines static strings for each HTTP method. For GET requests with the expected query string, the middleware function uses the `ContentType` property to set the `Content-Type` header and uses the `WriteAsync` method to add a string to the body of the response.

```
...
context.Response.ContentType = "text/plain";
await context.Response.WriteAsync("Custom Middleware \n");
...
```

Setting the `Content-Type` header is important because it prevents the subsequent middleware component from trying to set the response status code and headers. ASP.NET Core will always try to make sure that a valid HTTP response is sent, and this can lead to the response headers or status code being set after an earlier component has already written content to the response body, which produces an exception (because the headers have to be sent to the client before the response body can begin).

> **NOTE** In this part of the book, all the examples send simple string results to the browser. In part 3, I show you how to create web services that return JSON data and introduce the different ways that ASP.NET Core can produce HTML results.

The second argument to the middleware is the function conventionally named `next` that tells ASP.NET Core to pass the request to the next component in the request pipeline.

```
...
if (context.Request.Method == HttpMethods.Get
        && context.Request.Query["custom"] == "true") {
    context.Response.ContentType = "text/plain";
    await context.Response.WriteAsync("Custom Middleware \n");
}
await next();
...
```

No arguments are required when invoking the next middleware component because ASP.NET Core takes care of providing the component with the `HttpContext` object and its own `next` function so that it can process the request. The `next` function is asynchronous, which is why the `await` keyword is used and why the lambda function is defined with the `async` keyword.

TIP You may encounter middleware that calls `next.Invoke()` instead of `next()`. These are equivalent, and `next()` is provided as a convenience by the compiler to produce concise code.

Start ASP.NET Core using the `dotnet run` command and use a browser to request http://localhost:5000/?custom=true. You will see that the new middleware function writes its message to the response before passing on the request to the next middleware component, as shown in figure 12.5. Remove the query string, or change `true` to `false`, and the middleware component will pass on the request without adding to the response.

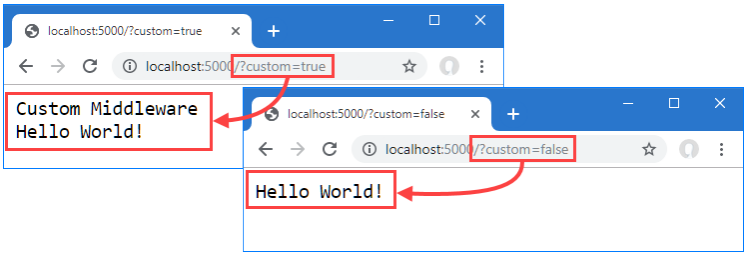

Figure 12.5 Creating custom middleware

12.4.1 Defining middleware using a class

Defining middleware using lambda functions is convenient, but it can lead to a long and complex series of statements in the `Program.cs` file and makes it hard to reuse middleware in different projects. Middleware can also be defined using classes, which keeps the code outside of the `Program.cs` file. To create a middleware class, add a class file named `Middleware.cs` to the `Platform` folder, with the content shown in listing 12.7.

Listing 12.7 The contents of the Middleware.cs file in the Platform folder

```
namespace Platform {

    public class QueryStringMiddleWare {
        private RequestDelegate next;

        public QueryStringMiddleWare(RequestDelegate nextDelegate) {
            next = nextDelegate;
        }

        public async Task Invoke(HttpContext context) {
            if (context.Request.Method == HttpMethods.Get
                        && context.Request.Query["custom"] == "true") {
                if (!context.Response.HasStarted) {
                    context.Response.ContentType = "text/plain";
                }
```

```
                await context.Response.WriteAsync("Class Middleware \n");
            }
            await next(context);
        }
    }
}
```

Middleware classes receive a `RequestDelegate` object as a constructor parameter, which is used to forward the request to the next component in the pipeline. The `Invoke` method is called by ASP.NET Core when a request is received and is given an `HttpContext` object that provides access to the request and response, using the same classes that lambda function middleware receives. The `RequestDelegate` returns a `Task`, which allows it to work asynchronously.

One important difference in class-based middleware is that the `HttpContext` object must be used as an argument when invoking the `RequestDelegate` to forward the request, like this:

```
...
await next(context);
...
```

Class-based middleware components are added to the pipeline with the `UseMiddleware` method, which accepts the middleware as a type argument, as shown in listing 12.8.

Listing 12.8 Adding class-based middleware in the Program.cs file in the Platform folder

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.Use(async (context, next) => {
    if (context.Request.Method == HttpMethods.Get
            && context.Request.Query["custom"] == "true") {
        context.Response.ContentType = "text/plain";
        await context.Response.WriteAsync("Custom Middleware \n");
    }
    await next();
});

app.UseMiddleware<Platform.QueryStringMiddleWare>();

app.MapGet("/", () => "Hello World!");

app.Run();
```

When the ASP.NET Core is started, the `QueryStringMiddleware` class will be instantiated, and its `Invoke` method will be called to process requests as they are received.

CAUTION A single middleware object is used to handle all requests, which means that the code in the `Invoke` method must be thread-safe.

Use the `dotnet run` command to start ASP.NET Core and use a browser to request http://localhost:5000/?custom=true. You will see the output from both middleware components, as shown in figure 12.6.

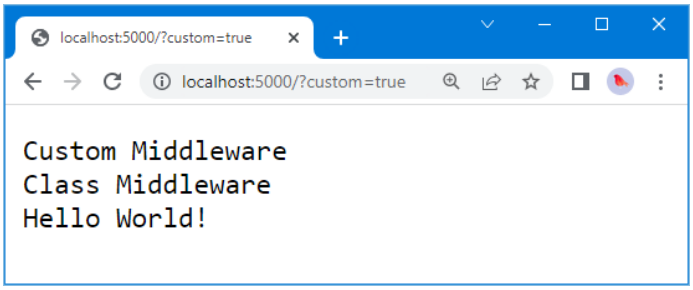

Figure 12.6 Using a class-based middleware component

12.4.2 Understanding the return pipeline path

Middleware components can modify the `HTTPResponse` object after the `next` function has been called, as shown by the new middleware in listing 12.9.

Listing 12.9 Adding new middleware in the Program.cs file in the Platform folder

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.Use(async (context, next) => {
    await next();
    await context.Response
        .WriteAsync($"\nStatus Code: { context.Response.StatusCode}");
});

app.Use(async (context, next) => {
    if (context.Request.Method == HttpMethods.Get
            && context.Request.Query["custom"] == "true") {
        context.Response.ContentType = "text/plain";
        await context.Response.WriteAsync("Custom Middleware \n");
    }
    await next();
});

app.UseMiddleware<Platform.QueryStringMiddleWare>();

app.MapGet("/", () => "Hello World!");

app.Run();
```

The new middleware immediately calls the `next` method to pass the request along the pipeline and then uses the `WriteAsync` method to add a string to the response body. This may seem like an odd approach, but it allows middleware to make changes to the response before and after it is passed along the request pipeline by defining statements before and after the next function is invoked, as illustrated by figure 12.7.

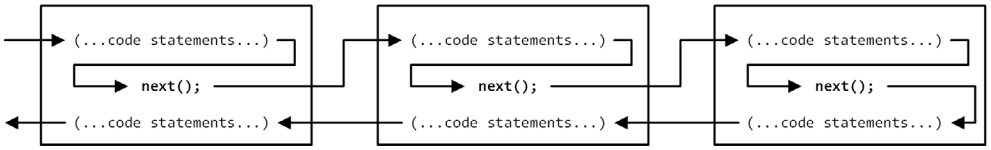

Figure 12.7 Passing requests and responses through the ASP.NET Core pipeline

Middleware can operate before the request is passed on, after the request has been processed by other components, or both. The result is that several middleware components collectively contribute to the response that is produced, each providing some aspect of the response or providing some feature or data that is used later in the pipeline.

Start ASP.NET Core using the `dotnet run` command and use a browser to request http://localhost:5000, which will produce output that includes the content from the new middleware component, as shown in figure 12.8.

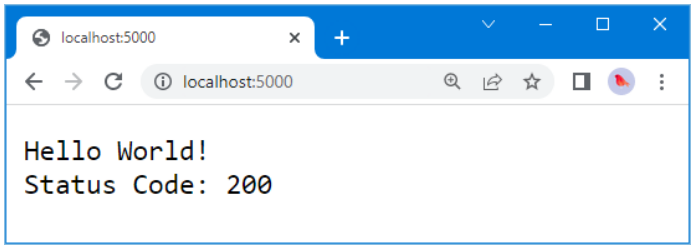

Figure 12.8 Modifying a response in the return path

> **NOTE** Middleware components must not change the response status code or headers once ASP.NET Core has started to send the response to the client. Check the `HasStarted` property, described in table 12.6, to avoid exceptions.

12.4.3 Short-Circuiting the request pipeline

Components that generate complete responses can choose not to call the `next` function so that the request isn't passed on. Components that don't pass on requests are said to *short-circuit* the pipeline, which is what the new middleware component shown in listing 12.10 does for requests that target the `/short` URL.

Listing 12.10 Short-Circuiting the pipeline in the Program.cs file in the Platform folder

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.Use(async (context, next) => {
    await next();
    await context.Response
```

```
        .WriteAsync($"\nStatus Code: { context.Response.StatusCode}");
});

app.Use(async (context, next) => {
    if (context.Request.Path == "/short") {
        await context.Response
            .WriteAsync($"Request Short Circuited");
    } else {
        await next();
    }
});

app.Use(async (context, next) => {
    if (context.Request.Method == HttpMethods.Get
            && context.Request.Query["custom"] == "true") {
        context.Response.ContentType = "text/plain";
        await context.Response.WriteAsync("Custom Middleware \n");
    }
    await next();
});

app.UseMiddleware<Platform.QueryStringMiddleWare>();

app.MapGet("/", () => "Hello World!");

app.Run();
```

The new middleware checks the `Path` property of the `HttpRequest` object to see whether the request is for the `/short` URL; if it is, it calls the `WriteAsync` method without calling the `next` function. To see the effect, restart ASP.NET Core and use a browser to request http://localhost:5000/short?custom=true, which will produce the output shown in figure 12.9.

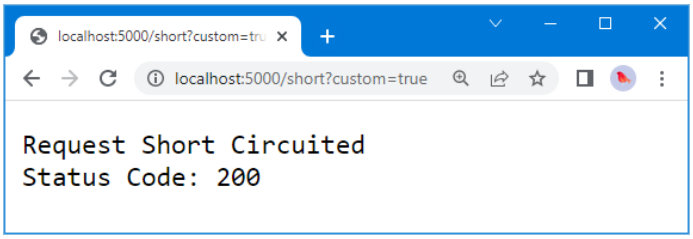

Figure 12.9 Short-circuiting the request pipeline

Even though the URL has the query string parameter that is expected by the next component in the pipeline, the request isn't forwarded, so that subsequent middleware doesn't get used. Notice, however, that the previous component in the pipeline has added its message to the response. That's because the short-circuiting only prevents components further along the pipeline from being used and doesn't affect earlier components, as illustrated in figure 12.10.

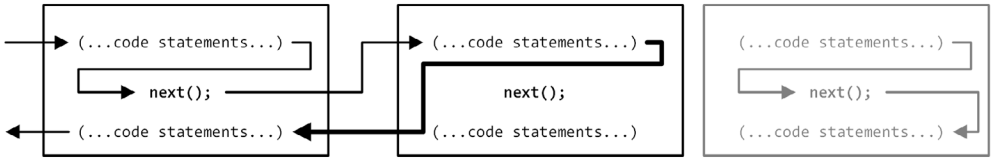

Figure 12.10 Short-circuiting the request pipeline

12.4.4 Creating pipeline branches

The `Map` method is used to create a section of pipeline that is used to process requests for specific URLs, creating a separate sequence of middleware components, as shown in listing 12.11.

Listing 12.11 Creating a pipeline branch in the Program.cs file in the Platform folder

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.Map("/branch", branch => {

    branch.UseMiddleware<Platform.QueryStringMiddleWare>();

    branch.Use(async (HttpContext context, Func<Task> next) => {
        await context.Response.WriteAsync($"Branch Middleware");
    });
});

app.UseMiddleware<Platform.QueryStringMiddleWare>();

app.MapGet("/", () => "Hello World!");

app.Run();
```

The first argument to the `Map` method specifies the string that will be used to match URLs. The second argument is the branch of the pipeline, to which middleware components are added with the `Use` and `UseMiddleware` methods.

The statements in listing 12.11 create a branch that is used for URLs that start with `/branch` and that pass requests through the `QueryStringMiddleware` class defined in listing 12.7 and a middleware lambda expression that adds a message to the response. Figure 12.11 shows the effect of the branch on the request pipeline.

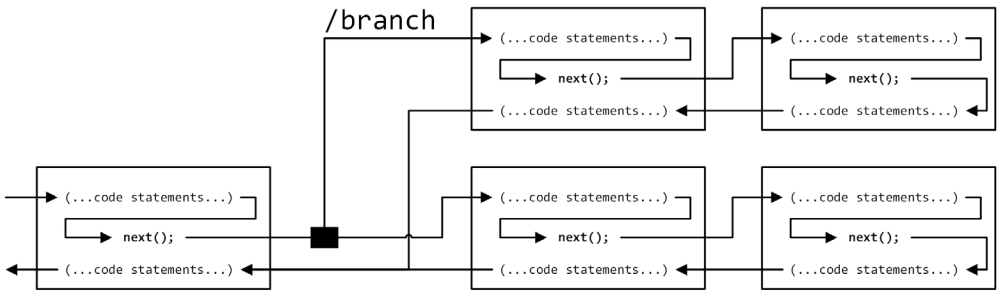

Figure 12.11 Adding a branch to the request pipeline

When a URL is matched by the `Map` method, it follows the branch. In this example, the final component in the middleware branch doesn't invoke the next delegate, which means that requests do not pass through the middleware components on the main path through the pipeline.

The same middleware can be used in different parts of the pipeline, which can be seen in listing 12.11, where the `QueryStringMiddleWare` class is used in both the main part of the pipeline and the branch.

To see the different ways that requests are handled, restart ASP.NET Core and use a browser to request the http://localhost:5000/?custom=true URL, which will be handled on the main part of the pipeline and will produce the output shown on the left of figure 12.12. Navigate to http://localhost:5000/branch?custom=true, and the request will be forwarded to the middleware in the branch, producing the output shown on the right in figure 12.12.

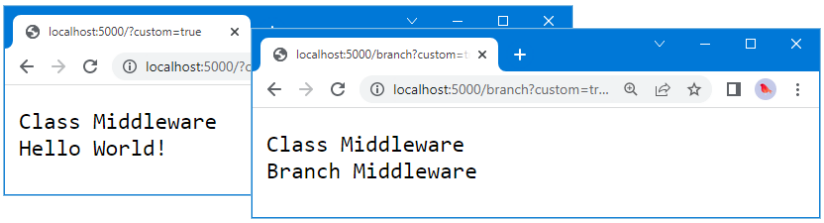

Figure 12.12 The effect of branching the request pipeline

Branching with a predicate

ASP.NET Core also supports the `MapWhen` method, which can match requests using a predicate, allowing requests to be selected for a pipeline branch on criteria other than just URLs.

The arguments to the `MapWhen` method are a predicate function that receives an `HttpContext` and that returns `true` for requests that should follow the branch, and a function that receives an `IApplicationBuilder` object representing the pipeline branch,

(continued)

to which middleware is added. Here is an example of using the `MapWhen` method to branch the pipeline:

```
...
app.MapWhen(context => context.Request.Query.Keys.Contains("branch"),
   branch => {
       // ...add middleware components here...
});
...
```

The predicate function returns `true` to branch for requests whose query string contains a parameter named `branch`. A cast to the `IApplicationBuilder` interface is not required because there only one `MapWhen` extension method has been defined.

12.4.5 Creating terminal middleware

Terminal middleware never forwards requests to other components and always marks the end of the request pipeline. There is a terminal middleware component in the `Program.cs` file, as shown here:

```
...
branch.Use(async (context, next) => {
    await context.Response.WriteAsync($"Branch Middleware");
});
...
```

ASP.NET Core supports the `Run` method as a convenience feature for creating terminal middleware, which makes it obvious that a middleware component won't forward requests and that a deliberate decision has been made not to call the `next` function. In listing 12.12, I have used the `Run` method for the terminal middleware in the pipeline branch.

Listing 12.12 Using the run method in the Program.cs file in the Platform folder

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

((IApplicationBuilder)app).Map("/branch", branch => {

    branch.UseMiddleware<Platform.QueryStringMiddleWare>();

    branch.Run(async (context) => {
        await context.Response.WriteAsync($"Branch Middleware");
    });
});

app.UseMiddleware<Platform.QueryStringMiddleWare>();

app.MapGet("/", () => "Hello World!");

app.Run();
```

The middleware function passed to the `Run` method receives only an `HttpContext` object and doesn't have to define a parameter that isn't used. Behind the scenes, the `Run` method is implemented through the `Use` method, and this feature is provided only as a convenience.

CAUTION Middleware added to the pipeline after a terminal component will never receive requests. ASP.NET Core won't warn you if you add a terminal component before the end of the pipeline.

Class-based components can be written so they can be used as both regular and terminal middleware, as shown in listing 12.13.

Listing 12.13 Adding terminal support in the Middleware.cs file in the Platform folder

```
namespace Platform {

    public class QueryStringMiddleWare {
        private RequestDelegate? next;

        public QueryStringMiddleWare() {
            // do nothing
        }

        public QueryStringMiddleWare(RequestDelegate nextDelegate) {
            next = nextDelegate;
        }

        public async Task Invoke(HttpContext context) {
            if (context.Request.Method == HttpMethods.Get
                        && context.Request.Query["custom"] == "true") {
                if (!context.Response.HasStarted) {
                    context.Response.ContentType = "text/plain";
                }
                await context.Response.WriteAsync("Class Middleware\n");
            }
            if (next != null) {
                await next(context);
            }
        }
    }
}
```

The component will forward requests only when the constructor has been provided with a non-`null` value for the `nextDelegate` parameter. listing 12.14 shows the application of the component in both standard and terminal forms.

Listing 12.14 Applying middleware in the Program.cs file in the Platform folder

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

((IApplicationBuilder)app).Map("/branch", branch => {
    branch.Run(new Platform.QueryStringMiddleWare().Invoke);
```

```
});

app.UseMiddleware<Platform.QueryStringMiddleWare>();

app.MapGet("/", () => "Hello World!");

app.Run();
```

There is no equivalent to the `UseMiddleware` method for terminal middleware, so the `Run` method must be used by creating a new instance of the middleware class and selecting its `Invoke` method. Using the `Run` method doesn't alter the output from the middleware, which you can see by restarting ASP.NET Core and navigating to the http://localhost:5000/branch?custom=true URL, which produces the content shown in figure 12.13.

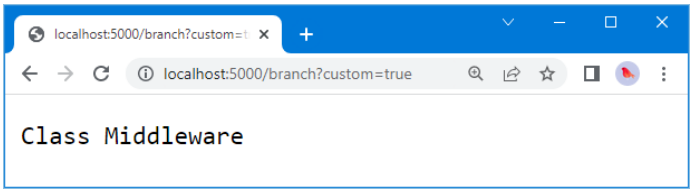

Figure 12.13. Using the Run method to create terminal middleware

12.5 Configuring middleware

There is a common pattern for configuring middleware that is known as the *options pattern* and that is used by some of the built-in middleware components described in later chapters.

The starting point is to define a class that contains the configuration options for a middleware component. Add a class file named `MessageOptions.cs` to the `Platform` folder with the code shown in listing 12.15.

Listing 12.15 The contents of the MessageOptions.cs file in the Platform folder

```
namespace Platform {

    public class MessageOptions {

        public string CityName { get; set; } = "New York";
        public string CountryName{ get; set; } = "USA";
    }
}
```

The `MessageOptions` class defines properties that detail a city and a country. In listing 12.16, I have used the options pattern to create a custom middleware component that relies on the `MessageOptions` class for its configuration. I have also removed some of the middleware from previous examples for brevity.

Listing 12.16 Using the options pattern in the Program.cs file in the Platform folder

```
using Microsoft.Extensions.Options;
using Platform;

var builder = WebApplication.CreateBuilder(args);

builder.Services.Configure<MessageOptions>(options => {
    options.CityName = "Albany";
});

var app = builder.Build();

app.MapGet("/location", async (HttpContext context,
    IOptions<MessageOptions> msgOpts) => {
        Platform.MessageOptions opts = msgOpts.Value;
        await context.Response.WriteAsync($"{opts.CityName}, "
            + opts.CountryName);
    });

app.MapGet("/", () => "Hello World!");

app.Run();
```

The options are set up using the `Services.Configure` method defined by the `WebApplicationBuilder` class, using a generic type parameter like this:

```
...
builder.Services.Configure<MessageOptions>(options => {
    options.CityName = "Albany";
});
...
```

This statement creates options using the `MessageOptions` class and changes the value of the `CityName` property. When the application starts, the ASP.NET Core platform will create a new instance of the `MessageOptions` class and pass it to the function supplied as the argument to the `Configure` method, allowing the default option values to be changed.

The options will be available as a service, which means this statement must appear before the call to the `Build` method is called, as shown in the listing.

Middleware components can access the configuration options by defining a parameter for the function that handles the request, like this:

```
...
app.MapGet("/location", async (HttpContext context,
    IOptions<MessageOptions> msgOpts) => {
        Platform.MessageOptions opts = msgOpts.Value;
        await context.Response.WriteAsync($"{opts.CityName}, "
            + opts.CountryName);
    });
...
```

Some of the extension methods used to register middleware components will accept any function to handle requests. When a request is processed, the ASP.NET

Core platform inspects the function to find parameters that require services, which allows the middleware component to use the configuration options in the response it generates:

```
...
app.MapGet("/location", async (HttpContext context,
    IOptions<MessageOptions> msgOpts) => {
        Platform.MessageOptions opts = msgOpts.Value;
        await context.Response.WriteAsync($"{opts.CityName}, "
            + opts.CountryName);
    });
...
```

This is an example of dependency injection, which I describe in detail in chapter 14. For now, however, you can see how the middleware component uses the options pattern by restarting ASP.NET Core and using a browser to request http://localhost:5000/location, which will produce the response shown in figure 12.14.

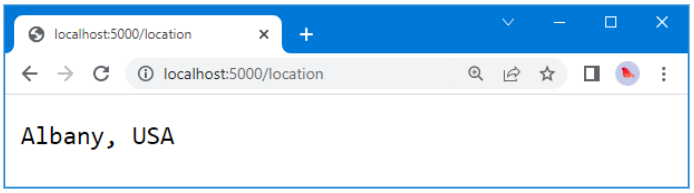

Figure 12.14 Using the options pattern

12.5.1 Using the options pattern with class-based middleware

The options pattern can also be used with class-based middleware and is applied in a similar way. Add the statements shown in listing 12.17 to the `Middleware.cs` file to define a class-based middleware component that uses the `MessageOptions` class for configuration.

Listing 12.17 Defining middleware in the Middleware.cs file in the Platform folder

```
using Microsoft.Extensions.Options;

namespace Platform {

    public class QueryStringMiddleWare {
        private RequestDelegate? next;

        // ...statements omitted for brevity...
    }

    public class LocationMiddleware {
        private RequestDelegate next;
        private MessageOptions options;

        public LocationMiddleware(RequestDelegate nextDelegate,
```

```
                IOptions<MessageOptions> opts) {
            next = nextDelegate;
            options = opts.Value;
        }

        public async Task Invoke(HttpContext context) {
            if (context.Request.Path == "/location") {
                await context.Response
                    .WriteAsync($"{options.CityName}, "
                        + options.CountryName);
            } else {
                await next(context);
            }
        }
    }

}
```

The `LocationMiddleware` class defines an `IOptions<MessageOptions>` constructor parameter, which can be used in the `Invoke` method to access the options settings.

Listing 12.18 reconfigures the request pipeline to replace the lambda function middleware component with the class from listing 12.17.

Listing 12.18 Using class-based middleware in the Program.cs file in the Platform folder

```
//using Microsoft.Extensions.Options;
using Platform;

var builder = WebApplication.CreateBuilder(args);

builder.Services.Configure<MessageOptions>(options => {
    options.CityName = "Albany";
});

var app = builder.Build();

app.UseMiddleware<LocationMiddleware>();

app.MapGet("/", () => "Hello World!");

app.Run();
```

When the `UseMiddleware` statement is executed, the `LocationMiddleware` constructor is inspected, and its `IOptions<MessageOptions>` parameter will be resolved using the object created with the `Services.Configure` method. This is done using the dependency injection feature that is described in chapter 14, but the immediate effect is that the options pattern can be used to easily configure class-based middleware. Restart ASP.NET Core and request http://localhost:5000/location to test the new middleware, which will produce the same output as shown in figure 12.14.

Summary

- ASP.NET Core uses a pipeline to process HTTP requests.
- Each request is passed to a series of middleware components for processing.
- Once the request has reached the end of the pipeline, the same middleware components are able to inspect and modify the response before it is sent.
- Middleware components can choose not to forward requests to the next component in the pipeline, known as "short-circuiting."
- ASP.NET Core can be configured to use different sequences of middleware components to handle different request URLs.
- Middleware is configured using the options pattern, which is a simple and consistent approach used throughout ASP.NET Core.